Improving search order for reachability testing in timed automata

Frédéric Herbreteau and Thanh-Tung Tran

Université de Bordeaux, Bordeaux INP, CNRS, LaBRI UMR5800
LaBRI Bât A30, 351 crs Libération, 33405 Talence, France

Abstract. Standard algorithms for reachability analysis of timed automata are sensitive to the order in which the transitions of the automata are taken. To tackle this problem, we propose a *ranking system* and a *waiting strategy*. This paper discusses the reason why the search order matters and shows how a ranking system and a waiting strategy can be integrated into the standard reachability algorithm to alleviate and prevent the problem respectively. Experiments show that the combination of the two approaches gives optimal search order on standard benchmarks except for one example. This suggests that it should be used instead of the standard BFS algorithm for reachability analysis of timed automata.

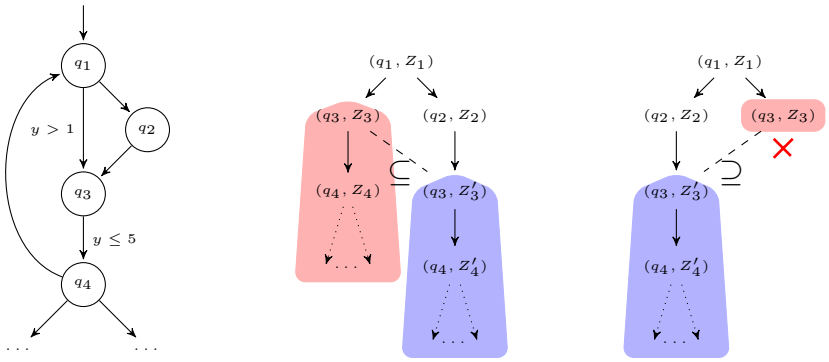
1 Introduction

Reachability analysis for timed automata asks if there is an execution of an automaton reaching a given state. This analysis can be used to verify all kinds of safety properties of timed systems. The standard approach to reachability analysis of timed automata uses sets of clock valuations, called *zones*, to reduce the reachability problem in the infinite state space of a timed automaton to the reachability problem in a finite graph. We present two heuristics to improve the efficiency of the zone based reachability algorithm.

The algorithm for reachability analysis of timed automata is a depth-first search, or a breadth-first search on a graph whose nodes are pairs consisting of a state of the automaton and a zone describing the set of possible clock valuations in this state. The use of zone inclusion is crucial for efficiency of this algorithm. It permits to stop exploration from a smaller zone if a bigger zone with the same state has been already explored.

Due to the use of zone inclusion the algorithm is sometimes very sensitive to exploration order. Indeed, it may happen that a small zone is reached and explored first, but then it is removed when a bigger zone is reached later. We will refer to such a situation as a *mistake*. A mistake can often be avoided by taking a different exploration order that reaches the bigger zone first.

In this paper we propose two heuristics to reduce the number of mistakes in the reachability analysis. In the example below we explain the mistake phenomenon in more details, and point out that it can cause an exponential blowup in the search space; this happens in the FDDI standard benchmark. The two



(a) Timed automaton. (b) In wrong order. (c) In good order.

Fig. 1: A timed automaton and two exploration graphs of its state-space. On the left, the transition to q_3 is explored first, which results in exploring the subtree of q_3 twice. On the right, the transition to q_2 is explored first and subsumption stops the second exploration as Z_3 is included in Z'_3.

heuristics are quite different in nature, so we evaluate their performance on the standard examples. Based on these experimental results we propose a simple modification to the standard exploration algorithm that significantly improves the exploration order.

We now give a concrete example showing why exploration order matters. Consider the timed automaton shown in Figure 1a, and assume we perform a depth-first search (DFS) exploration of its state space. The algorithm starts in (q_1, Z_1) where $Z_1 = (y \geq 0)$ is the set of all clock values. Assume that the transition to q_3 is taken first as in Figure 1b. The algorithm reaches the node (q_3, Z_3) with $Z_3 = (y > 1)$ and explores its entire subtree. Then, the algorithm backtracks to (q_1, Z_1) and proceeds with the transition to q_2 reaching (q_2, Z_2), and then (q_3, Z'_3) with $Z_2 = Z'_3 = (y \geq 0)$. It happens that $Z_3 \subseteq Z'_3$: the node (q_3, Z'_3) is *bigger* than the node (q_3, Z_3) which has been previously visited. At this point, the algorithm has to visit the entire subtree of (q_3, Z'_3) since the clock valuations in $Z'_3 \setminus Z_3$ have not been explored. The net result is that the earlier exploration from (q_3, Z_3) turns out to be useless since we need to explore from (q_3, Z'_3) anyway. If, by chance, our DFS exploration had taken different order of transitions, and first considered the one from q_1 to q_2 as in Figure 1c, the exploration would stop at (q_3, Z_3) since the bigger node (q_3, Z'_3) has already been visited and $Z_3 \subseteq Z'_3$. To sum up, in some cases DFS exploration is very sensible to the search order.

Several authors [3,6] have observed that BFS exploration is often much more efficient than DFS for reachability testing in timed automata. This can be attributed to an empirical observation that often a zone obtained by a short path is bigger than the one obtained by a longer path. This is the opposite in our example from Figure 1a. In consequence, a BFS algorithm will also do unnec-

essary explorations. When (q_3, Z_3') is visited, the node (q_4, Z_4) is already in the queue. Hence, while the algorithm has a chance to realise that exploring (q_3, Z_3) is useless due to the bigger node (q_3, Z_3'), it will keep visiting (q_4, Z_4) and all the subtree of (q_3, Z_3). Indeed, in the standard BFS algorithm, there is no mechanism to remove (q_4, Z_4) from the queue when (q_3, Z_3') is reached. Again, considering the transition from q_1 to q_2 before the transition to q_3 as in Figure 1c, avoids unnecessary exploration. Yet, by making the path $q_1 \to q_2 \to q_3$ one step longer we would obtain an example where all choices of search order would lead to unnecessary exploration. Overall, the standard reachability algorithm for timed automata, be it DFS or BFS, is sensitive to the alignment between the discovery of big nodes and the exploration of small nodes.

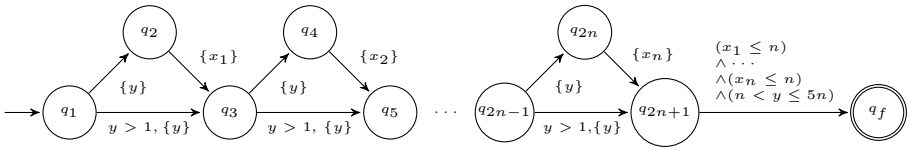


Fig. 2: Timed automaton with a racing situation.

One could ask what can be the impact of a pattern from Figure 1a, and does it really occur. The blowup of the exploration space can be exponential. One example is presented in Figure 2. It is obtained by iterating n times the pattern we have discussed above. The final state q_f is not reachable. By a similar analysis we can show that both the BFS and DFS algorithms with wrong exploration order explore and store exponentially more nodes than needed. In the automaton there are 2^n different paths to q_{2n+1}. The longest path $q_1, q_2, q_3, \ldots, q_{2n+1}$ generates the biggest zone, while there are about 2^n different zones that can be generated by taking different paths. If the DFS takes the worst exploration order, all these zones will be generated. If it takes the wrong order half of the times, then about $2^{n/2}$ zones will be generated. Similarly for BFS.

In the experiments section we show that, this far from optimal behaviour of BFS and DFS exploration indeed happens in the FDDI model, a standard benchmark model for timed automata.

In this paper we propose simple modifications of the exploration strategy to counter the problem as presented in the above examples. We will first describe a *ranking system* that mitigates the problem by assigning ranks to states, and using ranks to chose the transitions to explore. It will be rather clear that this system addresses the problem from our examples. Then we will propose *waiting strategy* that starts from a different point of view and is simpler to implement. The experiments on standard benchmarks show that the two approaches are incomparable but they can be combined to give optimal results in most of the cases. Since this combination is easy to implement, we propose to use it instead of standard BFS for reachability checking.

Related work: The influence of the search order has been discussed in the literature in the context of state-caching [7, 11–13], and state-space fragmentation [3,6,8]. State-caching focuses on limiting the number of stored nodes at the cost of exploring more nodes. We propose a strategy that improves the number of visited nodes as well as the number of stored nodes. In [3, 6, 8], it is suggested that BFS is the best search order to avoid state-space fragmentation in distributed model checking. We have not yet experimented our approach for distributed state-space exploration.

In terms of implementation, our approaches add a metric to states. In a different context a metric mechanism has been used by Behrmann *et al.* to guide the exploration in priced timed automata in [5].

Organisation of the paper: In the next section we present preliminaries for this paper: timed automata, the reachability problem and the standard reachability algorithm for timed automata. In Section 3, we propose a ranking system to limit the impact of mistakes during exploration. Section 4 presents another strategy that aims at limiting the number of mistakes. Finally, Section 5 gives some experimental results on the standard benchmarks.

2 Preliminaries

We introduce preliminary notions about timed automata and the reachability problem. Then, we introduce the classical zone-based algorithm used to solve this problem.

2.1 Timed Automata and the Reachability Problem

Let $X = \{x_1, \ldots, x_n\}$ be a set of clocks, i.e. variables that range over the non-negative real numbers $\mathbb{R}_{\geq 0}$. A *clock constraint* ϕ is a conjunction of constraints $x \# c$ for $x \in X$, $\# \in \{<, \leq, =, \geq, >\}$ and $c \in \mathbb{N}$. Let $\Phi(X)$ be the set of clock constraints over the set of clocks X. A *valuation* over X is a function $v : X \to \mathbb{R}_{\geq 0}$. We denote by $\mathbf{0}$ the valuation that maps each clock in X to 0, and by $\mathbb{R}_{\geq 0}^X$ the set of valuations over X. A valuation v satisfies a clock constraint $\phi \in \Phi(X)$, denoted $v \models \phi$, when all the constraints in ϕ hold after replacing every clock x by its value $v(x)$. For $\delta \in \mathbb{R}_{\geq 0}$, we denote $v + \delta$ the valuation that maps every clock x to $v(x) + \delta$. For $R \subseteq X$, $R[v]$ is the valuation that sets x to 0 if $x \in R$, and that sets x to $v(x)$ otherwise.

A *timed automaton (TA)* is a tuple $\mathcal{A} = (Q, q_0, F, X, Act, T)$ where Q is a finite set of states with initial state $q_0 \in Q$ and accepting states $F \subseteq Q$, X is a finite set of clocks, Act is a finite alphabet of actions, $T \subseteq Q \times \Phi(X) \times 2^X \times Act \times Q$ is a finite set of transitions (q, g, R, a, q') where g is a *guard*, R is the set of clocks that are *reset* and a is the *action* of the transition.

The semantics of a TA $\mathcal{A}$ is given by a transition system whose states are *configurations* $(q, v) \in Q \times \mathbb{R}_{\geq 0}^X$. The *initial configuration* is $(q_0, \mathbf{0})$. We have delay transitions: $(q, v) \xrightarrow{\delta} (q, v+\delta)$ for $\delta \in \mathbb{R}_{\geq 0}$, and action transitions: $(q, v) \xrightarrow{a} (q', v')$

if there exists a transition $(q, g, R, a, q') \in T$ such that $v \models g$ and $v' = [R]v$. A *run* is a finite sequence of transitions starting from the initial configuration $(q_0, \mathbf{0})$. A run is *accepting* is it ends in a configuration (q, v) with an accepting state $q \in F$.

The *reachability problem* consists in deciding if a given TA $\mathcal{A}$ has an accepting run. This problem is known to be PSPACE-complete [1].

2.2 Symbolic Semantics

The reachability problem cannot be solved directly from $\mathcal{A}$ due to the uncountable number of configurations. The standard solution is to use symbolic semantics of timed automata by grouping valuations together. A *zone* is a set of valuations described by a conjunction of two kinds of constraints: $x_i \# c$ and $x_i - x_j \# c$ where $x_i, x_j \in X$, $c \in \mathbb{Z}$ and $\# \in \{<, \leq, =, \geq, >\}$.

The *zone graph* $\mathsf{ZG}(\mathcal{A})$ of a timed automaton $\mathcal{A} = (Q, q_0, F, X, Act, T)$ is a transition system with nodes of the form (q, Z) where $q \in Q$ and Z is a zone. The initial node is (q_0, Z_0) where $Z_0 = \{\mathbf{0} + \delta \mid \delta \in \mathbb{R}_{\geq 0}\}$. The nodes (q, Z) with $q \in F$ are accepting. There is a transition $(q, Z) \Rightarrow (q', Z')$ if there exists a transition $(q, g, R, a, q') \in T$ such that $Z' = \{v' \in \mathbb{R}_{\geq 0} \mid \exists v \in Z\, \exists \delta \in \mathbb{R}_{\geq 0}\ (q, v) \xrightarrow{a} \xrightarrow{\delta} (q', v')\}$ and $Z' \neq \emptyset$. The relation $\Rightarrow$ is well-defined as it can be shown that if Z is a zone, then Z' is a zone. Zones can be efficiently represented by Difference Bound Matrices (DBMs) [10] and the successor Z' of a zone Z can be efficiently computed using this representation.

The zone graph $\mathsf{ZG}(\mathcal{A})$ is still infinite [9], and an additional abstraction step is needed to obtain a finite transition system. An *abstraction operator* is a function $\mathfrak{a} : \mathcal{P}(\mathbb{R}^X_{\geq 0}) \to \mathcal{P}(\mathbb{R}^X_{\geq 0})$ such that $W \subseteq \mathfrak{a}(W)$ and $\mathfrak{a}(\mathfrak{a}(W)) = \mathfrak{a}(W)$ for every set W of valuations. An abstraction operator defines an abstract symbolic semantics. Similarly to the zone graph, we define the *abstract zone graph* $\mathsf{ZG}^{\mathfrak{a}}(\mathcal{A})$. Its initial node is $(q_0, \mathfrak{a}(Z_0))$ and we have a transition $(q, Z) \Rightarrow_{\mathfrak{a}} (q', \mathfrak{a}(Z'))$ if $\mathfrak{a}(Z) = Z$ and $(q, Z) \Rightarrow (q', Z')$.

In order to solve the reachability problem for $\mathcal{A}$ from $\mathsf{ZG}^{\mathfrak{a}}(\mathcal{A})$, the abstraction operator $\mathfrak{a}$ should have the property that every run of $\mathcal{A}$ has a corresponding path in $\mathsf{ZG}^{\mathfrak{a}}(\mathcal{A})$ (completeness) and conversely, every path in $\mathsf{ZG}^{\mathfrak{a}}(\mathcal{A})$ should correspond to a run in $\mathcal{A}$ (soundness). Furthermore, $\mathsf{ZG}^{\mathfrak{a}}(\mathcal{A})$ should be finite. Several abstraction operators have been introduced in the literature [4, 9]. The abstraction operator $\mathsf{Extra_{LU}}^+$ [4] has all the required properties above. Moreover, the $\mathsf{Extra_{LU}}^+$ abstraction of a zone is itself a zone. It can be computed from the DBM representation of the zone. This allows to compute the abstract zone graph efficiently using DBMs as a symbolic representation for zones. The $\mathsf{Extra_{LU}}^+$ abstraction is used by most implementation including the state-of-the-art tool UPPAAL [2]. The theorem below reduces the reachability problem for $\mathcal{A}$ to the reachability problem in the finite graph $\mathsf{ZG}^{\mathsf{Extra_{LU}}^+}(\mathcal{A})$.

Theorem 1 ([4]). *There is an accepting run in $\mathcal{A}$ iff there exists a path in $\mathsf{ZG}^{\mathsf{Extra_{LU}}^+}(\mathcal{A})$ from $(q_0, \mathsf{Extra_{LU}}^+(Z_0))$ to some state (q, Z) with $q \in F$. Furthermore $\mathsf{ZG}^{\mathsf{Extra_{LU}}^+}(\mathcal{A})$ is finite.*

Algorithm 1.1: Standard reachability algorithm for timed automaton $\mathcal{A}$.

```
1  function reachability_check(A)
2    W := {(q0, Extra_LU+(Z0))}; P := W // Invariant: W ⊆ P
3
4    while (W ≠ ∅) do
5      take and remove a node (q, Z) from W
6      if (q is accepting)
7        return Yes
8      else
9        for each (q, Z) ⇒_Extra_LU+ (q', Z')
10         if there is no (qB, ZB) ∈ P s.t. (q', Z') ⊆ (qB, ZB)
11           for each (qS, ZS) ∈ P such that (qS, ZS) ⊆ (q', Z')
12             remove (qS, ZS) from W and P
13           add (q', Z') to W and to P
14
15   return No
```

2.3 Reachability algorithm

Algorithm 1.1 is the standard reachability algorithm for timed automata. It explores the finite abstract zone graph $\mathsf{ZG}^{\mathsf{Extra}_{\mathsf{LU}}{}^{+}}(\mathcal{A})$ of an automaton $\mathcal{A}$ from the initial node until it finds an accepting node, or it has visited the entire state-space of $\mathsf{ZG}^{\mathsf{Extra}_{\mathsf{LU}}{}^{+}}(\mathcal{A})$. It maintains a set of *waiting nodes* W and a set of *visited nodes* P such that $W \subseteq P$.

Algorithm 1.1 uses zone inclusion to stop exploration, and this is essential for its efficiency. We have $(q, Z) \subseteq (q', Z')$ when $q = q'$ and $Z \subseteq Z'$. Notice that zone inclusion is a simulation relation over nodes since zones are sets of valuations. Zone inclusion is first used in line 10 to stop the exploration in (q, Z) if there is a bigger node (q_B, Z_B) in P. It is also used in line 12 to only keep the maximal nodes w.r.t. $\subseteq$ in P and W.

Algorithm 1.1 does not specify any exploration strategy. As we have stressed in the introduction, the search order greatly influences the number of nodes visited by the algorithm and stored in the sets W and P. At first sight it may seem strange why there should be a big difference between, say, BFS and DFS search orders. The cause is the optimisation due to subsumption w.r.t. $\subseteq$ in lines 10 and 12. When equality on nodes is used instead of zone inclusion, every node is visited. Hence, BFS and DFS coincide in the sense that they will visit the same nodes, while not in the same order. The situation is very different with zone inclusion. Consider again the two nodes $(q_2, Z_2) \subseteq (q_2, Z_2')$ in Figure 1b. Since the smaller node (q_2, Z_2) is reached first, the entire subtrees of both nodes are visited whereas it would be sufficient to explore the subtree of the bigger node (q_2, Z_2') to solve the reachability problem. Indeed, every node below (q_2, Z_2) is simulated by the corresponding node below (q_2, Z_2'). Notice that the problem occurs both with a DFS and with a BFS strategy since the bigger node (q_2, Z_2')

is further from the root node than the smaller node (q_2, Z_2). When the bigger node is found before the smaller one, as in Figure 1c, only the subtree of the bigger node is visited. An optimal search strategy would guarantee that big nodes are visited before small ones. In the remaining of the paper we propose two heuristics to optimise the search order.

3 Ranking system

In this section we propose an exploration strategy to address the phenomenon we have presented in the introduction: we propose a solution to stop the exploration of the subtree of a small node when a bigger node is reached. As we have seen, the late discovery of big nodes causes unnecessary explorations of small nodes and their subtrees. In the worst case, the number of needlessly visited nodes may be exponential (cf. Figure 2).

Our goal is to minimise the number of visited nodes as well as the number of stored nodes (i.e. the size of P in Algorithm 1.1). Consider again the situation in Figure 1b where $(q_3, Z_3) \subseteq (q_3, Z'_3)$. When the big node (q_3, Z'_3) is reached, we learn that exploring the small node (q_3, Z_3) is unnecessary. In such a situation, Algorithm 1.1 erases the small node (q_3, Z_3) (line 10), but all its descendants that are in the waiting list W will be still explored.

A first and straightforward solution would be to erase the whole subtree of the small node (q_3, Z_3). Algorithm 1.1 would then proceed with the waiting nodes in the subtree of (q_3, Z'_3). This approach is however too rudimentary. Indeed, it may happen that the two nodes (q_4, Z_4) and (q_4, Z'_4) in Figure 1b are identical. Then, erasing the whole subtree of (q_3, Z_3) will lead to exploring (q_4, Z_4) and all its subtree twice. We have observed on classical benchmarks (see Section 5) that identical nodes are frequently found. While this approach is correct, it would result in visiting more nodes than the classical algorithm.

We propose a more subtle approach based on an interesting property of Algorithm 1.1. Consider the two nodes (q_4, Z_4) and (q_4, Z'_4) in Figure 1b again, and assume that (q_4, Z'_4) is reached after (q_4, Z_4). If the two nodes are identical, then (q_4, Z'_4) is erased by Algorithm 1.1 in line 10, but (q_4, Z_4) is kept since it has been visited first. Conversely, if the two nodes are different, we still have $(q_4, Z_4) \subseteq (q_4, Z'_4)$, then (q_4, Z_4) is erased by Algorithm 1.1 in line 10. Hence, as the algorithm explores the subtree of (q_3, Z'_3), it progressively erases all the nodes in the subtree of (q_3, Z_3) that are smaller than some node in the subtree of (q_3, Z'_3). At the same time, it keeps the nodes that are identical to some node below (q_3, Z'_3), hence avoiding several explorations of the same node.

Now, it remains to make all this happen before the subtree of (q_3, Z_3) is developed any further. This is achieved by giving a higher priority to (q_3, Z'_3) than all the waiting nodes below (q_3, Z_3). This priority mechanism is implemented by assigning a *rank* to every node.

Algorithm 1.2 below is a modified version of Algorithm 1.1 that implements the ranking of nodes (the modifications are highlighted). Nodes are initialised with rank 0. The rank of a node (q', Z') is updated with respect to the ranks

of the nodes (q_S, Z_S) that are simulated by (q', Z') (line 15). For each node (q_S, Z_S), we compute the maximum rank r of the waiting nodes below (q_S, Z_S). Then, $\mathsf{rank}(q', Z')$ is set to $\max(\mathsf{rank}(q', Z'), r+1)$ giving priority to (q', Z') over the waiting nodes below (q_S, Z_S).

Algorithm 1.2: Reachability algorithm with ranking of nodes for timed automaton $\mathcal{A}$. The set P is stored as a tree $\rightarrow$.

```
1  function reachability_check(A)
2    W := {(q0, Extra_LU+(Z0))}; P := W
3    init_rank(q0, Extra_LU+(Z0))
4
5    while (W ≠ ∅) do
6      take and remove a node (q, Z) with highest rank from W
7      if (q is accepting) then
8        return Yes
9      else
10       for each (q, Z) ⇒_{Extra_LU+} (q', Z')
11         init_rank(q', Z')
12         if there is no (qB, ZB) ∈ P s.t. (q', Z') ⊆ (qB, ZB) then
13           for each (qS, ZS) ∈ P s.t. (qS, ZS) ⊆ (q', Z')
14             if (qS, ZS) ∉ W then // implies not a leaf node in P
15               rank(q', Z') := max(rank(q', Z'), 1 + max_rank_waiting(qS, ZS))
16           remove (qS, ZS) from W and P
17         add (q', Z') to W and to P
18   return No
19
20 function max_rank_waiting(q, Z)
21   if (q, Z) is in W then // implies leaf node in P
22     return rank(q, Z)
23   else
24     r := 0;
25     for each edge (q, Z) → (q', Z') in P
26       r := max(r, max_rank_waiting(q', Z'))
27     return r
28
29 function init_rank(q, Z)
30   if Z is the true zone then
31     rank(q, Z) := ∞
32   else
33     rank(q, Z) := 0
```

The function max_rank_waiting determines the maximal rank among waiting nodes below (q_S, Z_S). To that purpose, the set of visited nodes P is stored as a reachability tree. When a node (q_S, Z_S) is removed in line 16, its parent node is connected to its child nodes to maintain reachability of waiting nodes. Observe that the node (q', Z') is added to the tree P in line 17 after its rank has been updated in line 15. This is needed in the particular case where (q_S, Z_S) is an ancestor of node (q', Z') in line 15. The rank of (q', Z') will be updated taking into account the waiting nodes below (q_S, Z_S). Obviously, (q', Z') should not be considered among those waiting nodes, which is guaranteed since (q', Z') does not belong to the tree yet.

The intuition behind the use of ranks suggest one more useful heuristic. Ranks are used to give priority to exploration from some nodes over the others. Nodes with true zones are a special case in this context, since they can never be covered, and in consequence it is always better to explore them first. We

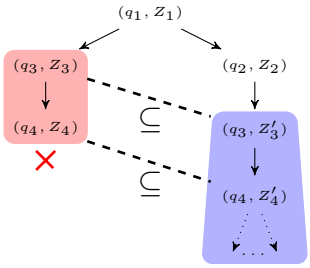


Fig. 3: Reachability tree for Algorithm 1.2 on the automaton in Figure 1a.

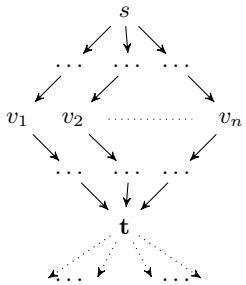


Fig. 4: Waiting strategy starts exploring from t only after all paths leading to t are explored.

implement this observation by simply assigning the biggest possible rank (∞) to such nodes (line 31 in the Algorithm).

Let us explain how the Algorithm 1.2 works on an example. Consider again the automaton in Figure 1a. The final exploration graph is depicted in Figure 3. When (q_1, Z_1) is visited, both (q_3, Z_3) and (q_2, Z_2) are put into the waiting list W with rank 0. Recall that exploring (q_3, Z_3) first is the worst exploration order. This adds (q_4, Z_4) to the waiting list with rank 0. The exploration of (q_2, Z_2) adds (q_3, Z'_3) to the waiting list. At this stage, the rank of (q_3, Z'_3) is set to 1 since it is bigger than (q_3, Z_3) which is erased. The node (q_3, Z'_3) has the highest priority among all waiting nodes and is explored next. This generates the node (q_4, Z'_4) that is bigger than (q_4, Z_4). Hence (q_4, Z_4) is erased, (q_4, Z'_4) gets rank 1 and the exploration proceeds from (q_4, Z'_4). One can see that, when a big node is reached, the algorithm not only stops the exploration of the smaller node but also of the nodes in its subtree. Figure 3 shows a clear improvement over Figure 1b.

4 Waiting strategy

We present an exploration strategy that will aim at reducing the number of exploration mistakes: situations when a bigger node is discovered later than a smaller one. The ranking strategy from the previous section reduced the cost of a mistake by stopping the exploration from descendants of a small node when it found a bigger node. By contrast, the waiting strategy of this section will not develop a node if it is aware of some other parts of exploration that may lead to a bigger node.

The waiting strategy is based on topological-like order on states of automata. We first present this strategy on a single automaton. Then we consider networks

of timed automata, and derive a topological-like ordering from orderings on the components. Before we start we explain what kind of phenomenon our strategy is capturing.

To see what we aim at, consider the part of a timed automaton depicted in Figure 4. There is a number of paths form state s to state t, not necessary of the same length. Suppose the search strategy from (s, Z) has reached (t, Z_1) by following the path through v_1. At this point it is reasonable to delay exploration from (t, Z_1) until all explorations of paths through $v_2, \ldots, v_k$ finish. This is because some of these explorations may result in a bigger zone than Z_1, and in consequence make an exploration from (t, Z_1) redundant.

The effect of such a waiting heuristic is clearly visible on our example from Figure 2. The automaton consists of segments: from q_1 to q_3, from q_3 to q_5, etc. Every segment is a very simple instance of the situation from Figure 4 that we have discussed in the last paragraph. There are two paths that lead from state q_1 to state q_3. These two paths have different lengths, so with a BFS exploration one of the paths will reach q_3 faster than the other. The longest path (that one going through q_2) gives the biggest zone in q_3; but BFS will no be able to use this information; and in consequence it will generate exponentially many nodes on this example. The waiting heuristic will collect all the search paths at states $q_3, q_5, \ldots$ and will explore only the best ones, so its search space will be linear.

We propose to implement these ideas via a simple modification of the standard algorithm. The waiting strategy will be based on a partial order $\sqsubseteq_{topo}$ of sates of $\mathcal{A}$. We will think of it as a topological order of the graph of the automaton (after removing cycles in some way). This order is then used to determine the exploration order.

Algorithm 1.3: Reachability algorithm with waiting strategy

This algorithm is obtained from the standard Algorithm 1.1 by changing line 5 to
take and **remove** (q, Z) minimal w.r.t. $\sqsubseteq_{topo}$ from W

In the remaining of the section we will propose some simple ways of finding a suitable $\sqsubseteq_{topo}$ order.

4.1 Topological-like ordering for a timed automaton

It is helpful to think of the order $\sqsubseteq_{topo}$ on states as some sort of topological ordering, but we cannot really assume this since the graphs of our automata may have loops. Given a timed automaton $\mathcal{A}$, we find a linear order on the states of $\mathcal{A}$ in two steps:

1. we find a maximal subset of transitions of $\mathcal{A}$ that gives a graph $\mathcal{A}_{DAG}$ without cycles;
2. then we compute a topological ordering of this graph.

Given an automaton $\mathcal{A}$, the graph $\mathcal{A}_{DAG}$ can be computed by running a depth-first search (DFS) from the initial state of $\mathcal{A}$. While traversing $\mathcal{A}$, we only

consider the transitions that point downwards or sideways; in other words we ignore all the transitions that lead to a state that is on the current search stack. At the end of the search, when all the states have been visited, the transitions that have not been ignored form a graph $\mathcal{A}_{DAG}$.

As an example, consider the timed automaton $\mathcal{A}$ in Figure 1a. The transition from q_4 to q_1 is ignored when computing $\mathcal{A}_{DAG}$ starting from q_1. A topological-like ordering is computed from the resulting graph: $q_1 \sqsubseteq_{topo} q_2 \sqsubseteq_{topo} q_3 \sqsubseteq_{topo} q_4$. Let us see how $\sqsubseteq_{topo}$ helps Algorithm 1.1 to explore bigger nodes first. Starting from node $(q_1, true)$, Algorithm 1.1 adds $(q_2, true)$ and $(q_3, y > 1)$ to the waiting list. Since $q_2 \sqsubseteq_{topo} q_3$, the algorithm then explores node $(q_2, true)$, hence adding node $(q_3, true)$ to the waiting list. The small node $(q_3, y > 1)$ is then automatically erased, and the exploration proceeds from the big node $(q_3, true)$. Observe that the exploration of the node $(q_3, y > 1)$ is postponed until the second path reaches q_3. Upon this stage, the zone inclusion relation will help to stop all explorations of smaller nodes; in our case it is $(q_3, y > 1)$. Thus, the algorithm performs optimally on this example, no exploration step can be avoided.

4.2 Topological-like ordering for networks of timed automata

Real-time systems often consist of several components that interact with each other. In order to apply the same approach we need to find an ordering on a set of global states of the system. For this we will find an ordering for each component and then extend it to the whole system without calculating the set of global states.

We suppose that each component of a system is modelled by a timed automaton $\mathcal{A}_i = (Q_i, q_{0_i}, F_i, X_i, Act_i, T_i)$. The system is modelled as the product $\mathcal{A} = (Q, q_0, F, X, Act, T)$ of the components $(\mathcal{A}_i)_{1 \leq i \leq k}$. The states of $\mathcal{A}$ are the tuples of states of $\mathcal{A}_1, \ldots, \mathcal{A}_k$: $Q = Q_1 \times \cdots \times Q_k$ with initial state $q_0 = \langle q_{0_1}, \ldots, q_{0_k} \rangle$ and final states $F = F_1 \times \cdots \times F_k$. Clocks and actions are shared among the processes: $X = \bigcup_{1 \leq i \leq k} X_i$ and $Act = \bigcup_{1 \leq i \leq k} Act_i$. Interactions are modelled by the synchronisation of processes over the same action. There is a transition $(\langle q_1, \ldots, q_n \rangle, g, R, a, \langle q'_1, \ldots, q'_n \rangle) \in T$ if

- either, there are two processes i and j with transitions $(q_i, g_i, R_i, \mathbf{a}, q'_i) \in T_i$ and $(q_j, g_j, R_j, \mathbf{a}, q'_j) \in T_j$ such that $g = g_i \wedge g_j$ and $R = R_i \cup R_j$, and $q'_l = q_l$ for every process $l \neq i, j$ (synchronised action)
- or there is a process i with transition $(q_i, g, R, a, q'_i) \in T_i$ such that for every process $l \neq i$, $a \notin Act_l$ and $q'_l = q_l$ (local action).

The product above allows synchronisation of 2 processes at a time. Our work does not rely on a specific synchronisation policy, hence other models of interactions (broadcast communications, n-ary synchronisation, etc.) could be considered as well. Notice that the product automaton $\mathcal{A}$ is, in general, exponentially bigger than each component $\mathcal{A}_i$.

The semantics of a network of timed automata $(\mathcal{A}_i)_{1 \leq i \leq k}$ is defined as the semantics of the corresponding product automaton $\mathcal{A}$. As a result, the reachability problem for $(\mathcal{A}_i)_{1 \leq i \leq k}$ reduces to the reachability problem in $\mathcal{A}$.

In order to apply the same approach as above, an ordering must be defined on the states of $\mathcal{A}$ which are tuples $\boldsymbol{q} = \langle q_1, \ldots, q_k \rangle$ of states of the component automata $\mathcal{A}_i$. It would not be reasonable to compute the product automaton $\mathcal{A}$ as its size grows exponentially with the number of its components. We propose an alternative solution that consists in computing a topological-like ordering $\sqsubseteq^i_{topo}$ for each component $\mathcal{A}_i$. To that purpose, we can apply the algorithm introduced in the previous section. Then, the ordering of tuples of states is defined pointwise:

Definition 1 (Joint ordering). *For $\boldsymbol{q}, \boldsymbol{q'} \in Q_1 \times \cdots \times Q_k$, we have $\boldsymbol{q} \sqsubseteq_{topo} \boldsymbol{q'}$ if $\boldsymbol{q}_i \sqsubseteq^i_{topo} \boldsymbol{q'}_i$ for all $1 \leq i \leq k$.*

Thus for networks of timed automata we consider the joint ordering in our waiting strategy.

5 Experimental evaluation

We present and comment the experimental results that we have performed. The results indicate that a mix of a ranking and waiting strategies avoids mistakes in most the examples.

We have evaluated the ranking system (Section 3) and the waiting strategy (Section 4) on classical benchmarks from the literature[1]: Critical Region (CR), Csma/Cd (C), Fddi (FD), Fischer (Fi), Flexray (Fl-PL) and Lynch (L), and on the BlowUp (B) example in Figure 2. These automata have no reachable accepting state, hence forcing algorithms to visit the entire state-space of the automata to prove unreachability.

Our objective is to avoid mistakes during exploration of the state-space of timed automata. At the end of the run of the algorithm, the set of visited nodes P forms an invariant showing that accepting nodes are unreachable. Every node that is visited by the algorithm and that does not belong to P at the end of the run is useless to prove unreachability. This happens when the algorithm does a mistake: it first visits a small node before reaching a bigger node later. We aim at finding a search order that visits bigger nodes first, hence doing as few mistakes as possible. Notice that it is not always possible to completely avoid mistakes since the only paths to a big node may have to visit a small node first.

We compare three algorithms in Table 1: BFS the standard breadth-first search algorithm[2] (i.e. Algorithm 1.1), R-BFS which implements a breadth-first search with priority to the highest ranked nodes (i.e. Algorithm 1.2) and TW-BFS which combines giving highest priority to true-zone nodes and the waiting strategy. We report on the number of visited nodes, the number of mistakes, the maximum number of stored nodes, and the final number of stored nodes. We also mention in column "visited ranking" the number of nodes that are re-visited to update the rank of the nodes by algorithm R-BFS (line 15 in Algorithm 1.2). The number of visited nodes gives a good estimate of the running time of the

[1] The models are available from http://www.labri.fr/perso/herbrete/tchecker.
[2] Algorithm 1.1 is essentially the algorithm that is implemented in UPPAAL [2].

algorithm, while the maximal number of stored nodes gives a precise indication of the memory used for the set P.

The ranking system gives very good results on all models except CSMA/CD. It makes no mistakes on FISCHER and LYNCH. This is due to the highest priority given to true-zone nodes. Indeed, column "visited ranking" shows that ranks are never updated, hence the nodes keep their initial rank. It also performs impressively well on BLOWUP, FDDI and FLEXRAY, gaining several orders of magnitude in the number of mistakes. However, it makes more mistakes than BFS on CSMA/CD. Indeed, the ranking system is efficient when big nodes are reached quickly, as the example in Figure 3 shows. When the big node (q_3, Z'_3) is reached, the ranking system stops the exploration of the subtree of the small node (q_3, Z_3) at (q_4, Z_4). However, making the path $q_1 \to q_2 \to q_3$ longer in the automaton in Figure 1a leads to explore a bigger part of the subtree of (q_3, Z_3). If this path is long enough, the entire subtree of (q_3, Z_3) may be visited before (q_3, Z'_3) is reached. The ranking system does not provide any help in this situation. This bad scenario occurs in the CSMA/CD example.

We have experimented the waiting strategy separately (not reported in Table 1). While the results are good on some models (BLOWUP, FDDI, CSMA/CD), the waiting strategy makes a lot more mistakes than the standard BFS on LYNCH and FLEXRAY. Indeed, the waiting strategy is sensitive to the topological ordering. Consider the automaton in Figure 1a with an extra transition $q_3 \to q_2$. The loop on q_2 and q_3 may lead to different topological orderings, for instance $q_1 \sqsubseteq_{topo} q_2 \sqsubseteq_{topo} q_3 \sqsubseteq_{topo} q_4$ and $q_1 \sqsubseteq_{topo} q_3 \sqsubseteq_{topo} q_2 \sqsubseteq_{topo} q_4$. These two choices lead to very different behaviours of the algorithm. Once the initial node has been explored, the two nodes $(q_3, y > 1)$ and $(q_2, true)$ are in the waiting queue. With the first ordering, $(q_2, true)$ is selected first and generates $(q_3, true)$ that cuts the exploration of the smaller node $(q_3, y > 1)$. However, with the second ordering $(q_3, y > 1)$ is visited first. As a result, $(q_3, true)$ is reached too late, and the entire subtree of $(q_3, y > 1)$ is explored unnecessarily. We have investigated the robustness of the waiting strategy w.r.t. random topological orderings for the models in Table 1. The experiments confirm that the waiting strategy is sensitive to topological ordering. For most models, the best results are achieved using the topological ordering that comes from running a DFS on the automaton as suggested in Section 4.1.

The two heuristics perform well on different models. This suggests to combine their strengths. Consider again the automaton in Figure 1a with an extra transition $q_3 \to q_2$. As explained above, due to the cycle on q_2 and q_3, several topological orderings are possible for the waiting strategy. The choice of $q_1 \sqsubseteq_{topo} q_3 \sqsubseteq_{topo} q_2 \sqsubseteq_{topo} q_4$ leads to a bad situation where $(q_3, y > 1)$ is taken first when the two nodes $(q_3, y > 1)$ and $(q_2, true)$ are in the waiting queue. As a result, the node $(q_3, y > 1)$ is visited without waiting the bigger node $(q_3, true)$. In such a situation, combining ranking and the waiting strategies helps. Indeed, after $(q_3, y > 1)$ has been explored, the waiting queue contains two nodes $(q_2, true)$ and $(q_4, 1 < y \leq 5)$. Since $q_2 \sqsubseteq_{topo} q_4$, the algorithm picks $(q_2, true)$,

hence generating $(q_3, true)$. As a true-zone node, $(q_3, true)$ immediately gets a higher rank than every waiting node. Exploring $(q_3, true)$ generates $(q_4, y \leq 5)$ that cuts the exploration from the small node $(q_4, 1 < y \leq 5)$.

We have tried several combinations of the two heuristics. The best one consists in using the waiting strategy with priority to true zones. More precisely, the resulting algorithm TW-BFS selects a waiting node as follows:

- True-zone nodes are taken in priority,
- If there is no true-zone node, the nodes are taken according to the waiting strategy, and in BFS order.

As Table 1 shows, TW-BFS makes no mistake on all models but three. Critical Region has unavoidable mistakes: big nodes that can only be reached after visiting a smaller node. The topological ordering used for Fddi is not optimal. Indeed, there exists an optimal topological search order for which TW-BFS makes no mistake, but it is not the one obtained by the algorithm presented in Section 4.1. Finally, the algorithm makes a lot of mistakes on Flexray, but the memory usage is almost optimal: the mistakes are quickly eliminated. This example is the only one where applying the ranking heuristic clearly outperforms TW-BFS.

We have also evaluated TW-BFS using randomised versions of the models in Table 1. Randomisation consists in taking the transitions in a non-fixed order, hence increasing the possibility of racing situations like in Figure 1. The experiments show that the strategies are robust to such randomisation, and the results on random instances are very close to the ones reported in the table.

The ranking strategy R-BFS requires to keep a tree structure over the passed nodes. Using the classical left child-right sibling encoding, the tree can be represented with only two pointers per node. This tree is explored when the rank of a node is updated (line 15 in Algorithm 1.2). Column "visited ranking" in Table 1 shows that these explorations do not inflict any significant overhead in terms of explored nodes, except for Csma/Cd and Critical Region for which it has been noticed above that algorithm R-BFS does not perform well. Furthermore, exploring the tree is inexpensive since the visited nodes, in particular the zones, have already been computed. Both the ranking strategy and the waiting strategy require to sort the list of waiting nodes. Our prototype implementation based on insertion sort is slow. However, preliminary experiments show that implementing the list of waiting nodes as a heap turns out to be very efficient.

To summarise we can consider our findings from a practical point of view of an implementation. The simplest to implement strategy would be to give priority to true zones. This would already give some improvements, but for example for Fddi there would be no improvement since there are no true zones. R-BFS gives very good results on Flexray model its implementation is more complex than TW-BFS strategy is relatively easy to implement and has very good performance on all but one model, where it is comparable to standard BFS. This suggests that TW-BFS could be used as a replacement for BFS.

	BFS				R-BFS					TW-BFS			
	visited	mist.	stored final	stored max	visited	mist.	stored final	stored max	visited ranking	visited	mist.	stored final	stored max
B-5	63	52	11	22	16	5	11	11	13	11	0	11	11
B-10	1254	1233	21	250	31	10	21	21	28	21	0	21	21
B-15	37091	37060	31	6125	46	15	31	31	43	31	0	31	31
FD-8	2635	2294	341	439	437	96	341	341	579	349	8	341	341
FD-10	10219	9694	525	999	684	159	525	525	1168	535	10	525	525
FD-15	320068	318908	1160	18707	1586	426	1160	1160	4543	1175	15	1160	1160
C-10	39698	5404	34294	48286	59371	25077	34294	52210	54319	34294	0	34294	34302
C-11	98118	17233	80885	124220	153042	72157	80885	130557	160822	80885	0	80885	80894
C-12	239128	50724	188404	311879	378493	190089	188404	320181	430125	188404	0	188404	188414
Fi-7	11951	4214	7737	7738	7737	0	7737	7737	0	7737	0	7737	7737
Fi-8	40536	15456	25080	25082	25080	0	25080	25080	0	25080	0	25080	25080
Fi-9	135485	54450	81035	81038	81035	0	81035	81035	0	81035	0	81035	81035
L-8	45656	15456	30200	30202	30200	0	30200	30200	0	30200	0	30200	30200
L-9	147005	54450	92555	92558	92555	0	92555	92555	0	92555	0	92555	92555
L-10	473198	186600	286598	286602	286598	0	286598	286598	0	286598	0	286598	286598
CR-3	1670	447	1223	1223	1532	309	1223	1223	1837	1563	340	1223	1223
CR-4	21180	7440	13740	13740	17694	3954	13740	13740	24295	19489	5749	13740	13740
CR-5	285094	113727	171367	171367	216957	45590	171367	171367	307010	257137	85770	171367	171367
Fl-PL	881214	228265	652949	652949	655653	2704	652949	652949	6977	12660557	11997402	663155	684467

Table 1: Experimental results: BFS corresponds to Algorithm 1.1 with a BFS order on the waiting nodes, R-BFS implements the ranking system on top of the BFS algorithm (i.e. Algorithm 1.2), and TW-BFS implements the waiting strategy with a priority to true-zone nodes.

6 Conclusion

We have analysed the phenomenon of mistakes in the zone based reachability algorithm for timed automata. This situation occurs when the exploration algorithm visits a node that is later removed due to a discovery of a bigger node. It is well known that DFS exploration may suffer from an important number of mistakes. We have exhibited examples where BFS makes an important number of mistakes that can be avoided.

To limit the number of mistakes in exploration we have proposed two heuristics: *ranking system* and the *waiting strategy*. The experiments on standard models show that, compared with the standard BFS reachability algorithm the strategies using our heuristics give not only a smaller number of visited nodes, but also a smaller number of stored nodes. Actually, on most examples our strategies are optimal as they do not make any mistakes. In addition, the experiments indicate that the TW-BFS strategy works often as good as the combination of both waiting and ranking strategies, while its implementation is much simpler. Therefore, we suggest to use the TW-BFS algorithm instead of standard BFS for reachability checking.

Acknowledgements. The authors wish to thank Igor Walukiewicz for the many helpful discussions.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical computer science 126(2), 183–235 (1994)

2. Behrmann, G., David, A., Larsen, K., Haakansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: QEST. pp. 125–126. IEEE Computer Society (2006)
3. Behrmann, G.: Distributed reachability analysis in timed automata. International Journal on Software Tools for Technology Transfer 7(1), 19–30 (2005)
4. Behrmann, G., Bouyer, P., Larsen, K.G., Pelánek, R.: Lower and upper bounds in zone-based abstractions of timed automata. International Journal on Software Tools for Technology Transfer 8(3), 204–215 (2006)
5. Behrmann, G., Fehnker, A.: Efficient guiding towards cost-optimality in UPPAAL. In: TACAS. pp. 174–188 (2001)
6. Behrmann, G., Hune, T., Vaandrager, F.W.: Distributing timed model checking - how the search order matters. In: CAV. pp. 216–231 (2000)
7. Behrmann, G., Larsen, K.G., Pelánek, R.: To store or not to store. In: CAV. pp. 433–445 (2003)
8. Braberman, V., Olivero, A., Schapachnik, F.: Zeus: A distributed timed model-checker based on Kronos. Electronic notes in theoretical computer science 68(4), 503–522 (2002)
9. Daws, C., Tripakis, S.: Model checking of real-time reachability properties using abstractions. In: TACAS, pp. 313–329. Springer (1998)
10. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: Automatic verification methods for finite state systems. pp. 197–212. Springer (1989)
11. Evangelista, S., Kristensen, L.: Search-order independent state caching. T. Petri Nets and Other Models of Concurrency 4, 21–41 (2010)
12. Godefroid, P., Holzmann, G.J., Pirottin, D.: State-space caching revisited. Formal Methods in System Design 7(3), 227–241 (1995)
13. Pleánek, R., Rosecký, V., Sedenka, J.: Evaluation of state caching and state compression techniques. Tech. rep., Masaryk University, Brno (2008)